Visual C and C++ (General) Technical Articles

C++: Under the Hood

Jan Gray

March 1994

Jan Gray is a Software Design Engineer in Microsoft's Visual C++ Business Unit. He helped design and implement the Microsoft Visual C++ compiler.

Introduction

It is important to understand how your programming language is implemented. Such knowledge dispels the fear and wonder of "What on earth is the compiler doing here?"; imparts confidence to use the new features; and provides insight when debugging and learning other language features. It also gives a feel for the relative costs of different coding choices that is necessary to write the most efficient code day to day.

This paper looks "under the hood" of C++, explaining "run-time" C++ implementation details such as class layout techniques and the virtual function call mechanism. Questions to be answered include:

- How are classes laid out?
- How are data members accessed?
- How are member functions called?
- What is an adjuster thunk?
- What are the costs:
  - Of single, multiple, and virtual inheritance?
  - Of virtual functions and virtual function calls?
  - Of casts to bases, to virtual bases?
  - Of exception handling?

First, we'll look at struct layout of C-like structs, single inheritance, multiple inheritance, and virtual inheritance, then consider data member access and member functions, virtual and not. We'll examine the workings of constructors, destructors, and assignment operator special member functions and dynamic construction and destruction of arrays. Finally, we'll briefly consider the impact of exception-handling support.

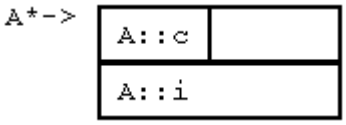
For each language feature topic, we'll very briefly present motivation and semantics for the language feature (although "Introduction to C++" this is not), and examine how the language feature was implemented in Microsoft® Visual C++™. Note the distinction between abstract language semantics and a particular concrete implementation. Other vendors have sometimes made different implementation choices for whatever reasons. In a few cases we contrast the Visual C++ implementation with others.

Class Layout

In this section we'll consider the storage layouts required for different kinds of inheritance.

C-like Structs

As C++ is based upon C, it is "mostly" upwards-compatible with C. In particular, the working papers specify the same simple struct layout rules that C has: Members are laid out in their declaration order, subject to implementation defined alignment padding. All C/C++ vendors ensure that valid C structs are stored identically by their C and C++ compilers. Here A is a simple C struct with the obvious expected member layout and padding.




```
struct A {
   char c;
   int i;
};
```

C-like Structs with C++ Features

Of course, C++ is an object-oriented programming language: It provides inheritance, encapsulation, and polymorphism by extending the mundane C struct into the wondrous C++ class. Besides data members, C++ classes can also encapsulate member functions and many other things. However, except for hidden data members introduced to implement virtual functions and virtual inheritance, the instance size is solely determined

by a class's data members and base classes.

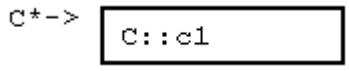
Here B is a C-like struct with some C++ features: There are public/protected/private access control declarations, member functions, static members, and nested type declarations. Only the non-virtual data members occupy space in each instance. Note that the standards committee working papers permit implementations to reorder data members separated by an access declarator, so these three members could have been laid out in any order. (In Visual C++, members are always laid out in declaration order, just as if they were members of a C struct)


```
struct B {
public:
   int bm1;
protected:
   int bm2;
private:
   int bm3;
   static int bsm;
   void bf();
   static void bsf();
   typedef void* bpv;
   struct N { };
};
```

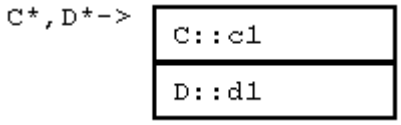
Single Inheritance

C++ provides inheritance to factor out and share common aspects of different types. A good example of a classes-with-inheritance data type organization is biology's classification of living things into kingdoms, phyla, orders, families, genus, species, and so on. This organization makes it possible to specify attributes, such as "mammals bear live young" at the most appropriate level of classification; these attributes are then *inherited* by other classes, so we can conclude without further specification that whales, squirrels, and people bear live young. Exceptional cases, such as platypi (a mammal, yet lays eggs), will require that we *override* the inherited attribute or behavior with one more appropriate for the derived class. More on that later.

In C++, inheritance is specified by using the "`: base`" syntax when defining the derived class. Here `D` is derived from its base class `C`.




```
struct C {
   int c1;
   void cf();
};
```




```
struct D : C {
   int d1;
   void df();
};
```

Since a derived class inherits all the properties and behavior of its base class, each instance of the derived class will contain a complete copy of the instance data of the base class. Within `D`, there is no requirement that `C`'s instance data precede `D`'s. But by laying `D` out this way, we ensure that the address of the `C` object within `D` corresponds to the address of the first byte of the `D` object. As we shall see, this eliminates adding a displacement to a `D*` when we need to obtain the address of its embedded `C`. This layout is used by all known C++ implementations. Thus, in a single inheritance class hierarchy, new instance data introduced in each derived class is simply appended to the layout of the base class. Note our layout diagram labels the "address points" of pointers to the `C` and `D` objects within a `D`.

Multiple Inheritance

Single inheritance is quite versatile and powerful, and generally adequate for expressing the (typically limited) degree of inheritance present in most design problems. Sometimes, however, we have two or more sets of behavior that we wish our derived class to acquire. C++ provides multiple inheritance to combine them.

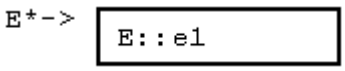
For instance, say we have a model for an organization that has a class `Manager` (who delegates) and class `Worker` (who actually does the work). Now how can we model a class `MiddleManager`, who, like a `Worker`, accepts work assignments from his/her manager and who, like a `Manager`, delegates this work to his/her employees? This is awkward to express using single inheritance: For `MiddleManager` to inherit behavior from both `Manager` and `Worker`, both must be base classes. If this is arranged so that `MiddleManager` inherits from `Manager` which inherits from `Worker`, it erroneously ascribes `Worker` behavior to `Manager`s. (Vice versa, the

same problem.) Of course, `MiddleManager` could inherit from just one (or neither) of `Worker` or `Manager`, and instead, duplicate (redeclare) both interfaces, but that defeats polymorphism, fails to reuse the existing interface, and leads to maintenance woes as interfaces evolve over time.

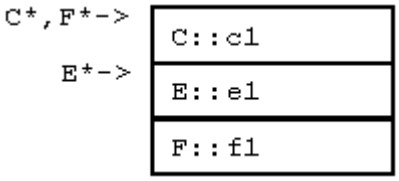
Instead, C++ allows a class to inherit from multiple base classes:

```
struct Manager ... { ... };
struct Worker ... { ... };
struct MiddleManager : Manager, Worker { ... };
```

How might this be represented? Continuing with our “classes of the alphabet” example:




```
struct E {
   int e1;
   void ef();
};
```




```
struct F : C, E {
   int f1;
   void ff();


};
```

Struct `F` multiply inherits from `C` and `E`. As with single inheritance, `F` contains a copy of the instance data of each of its base classes. Unlike single inheritance, it is not possible to make the address point of each bases’ embedded instance correspond to the address of the derived class:

```
F f;
// (void*)&f == (void*)(C*)&f;
// (void*)&f <  (void*)(E*)&f;
```

Here, the address point of the embedded `E` within `F` is not at the address of the `F` itself. As we shall see when we consider casts and member functions, this displacement leads to a small overhead that single inheritance does not generally require.

An implementation is free to lay out the various embedded base instances and the new instance data in any order. Visual C++ is typical in laying out the base instances in declaration order, followed by the new data members, also in declaration order. (As we shall see, this is not necessarily the case when some bases have virtual functions and others don't).

Virtual Inheritance

Returning to the `MiddleManager` example which motivated multiple inheritance in the first place, we have a problem. What if both `Manager` and `Worker` are derived from `Employee`?

```
struct Employee { ... };
struct Manager : Employee { ... };
struct Worker : Employee { ... };
struct MiddleManager : Manager, Worker { ... };
```

Since both `Worker` and `Manager` inherit from `Employee`, they each contain a copy of the `Employee` instance data. Unless something is done, each `MiddleManager` will contain two instances of `Employee`, one from each base. If `Employee` is a large object, this duplication may represent an unacceptable storage overhead. More seriously, the two copies of the `Employee` instance might get modified separately or inconsistently. We need a way to declare that `Manager` and `Worker` are each willing to share a single embedded instance of their `Employee` base class, should `Manager` or `Worker` ever be inherited with some other class that also wishes to share the `Employee` base class.

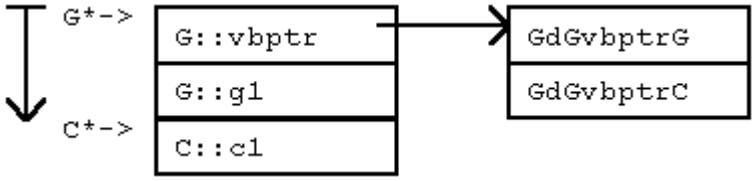
In C++, this "sharing inheritance" is (unfortunately) called virtual inheritance and is indicated by specifying that a base class is virtual.

```
struct Employee { ... };
struct Manager : virtual Employee { ... };
struct Worker : virtual Employee { ... };
struct MiddleManager : Manager, Worker { ... };
```

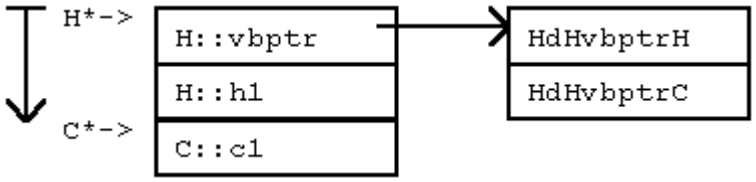
Virtual inheritance is considerably more expensive to implement and use than single and multiple inheritance. Recall that for single (and multiple) inherited bases and derived classes, the embedded base instances and their derived classes either share a common address point (as with single inheritance and the leftmost base inherited via multiple inheritance), or have a simple constant displacement to the embedded base instance (as

with multiple inherited non-leftmost bases, such as E). With virtual inheritance, on the other hand, there can (in general) be no fixed displacement from the address point of the derived class to its virtual base. If such a derived class is further derived from, the further deriving class may have to place the one shared copy of the virtual base at some other, different offset in the further derived class. Consider this example:

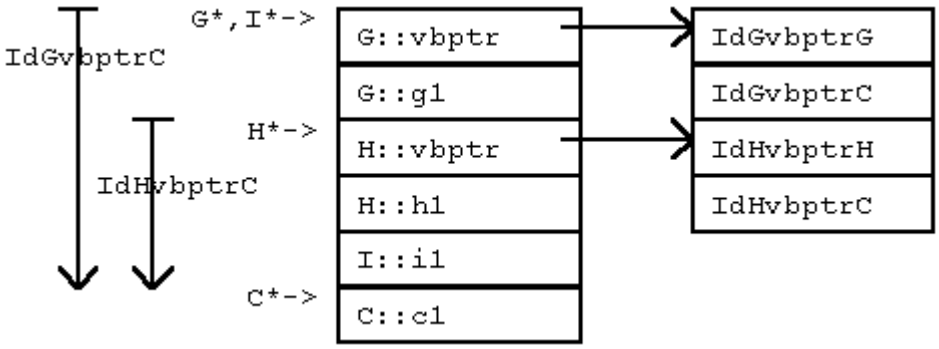
```
struct G : virtual C {
   int g1;
   void gf();
};
```




```
struct H : virtual C {
   int h1;
   void hf();
};
```




```
struct I : G, H {
   int i1;
   void _if();
};
```



Ignoring the `vbptr` members for a moment, notice that within a `G` object, the embedded `C` immediately follows the `G` data member, and similarly notice that within an `H`, the embedded `C` immediately follows the `H` data member. Now when we layout `I`, we can't preserve both relationships. In the Visual C++ layout above, the displacements from `G` to `C` in a `G` instance and in an `I` instance are different. Since classes are generally compiled without knowledge of how they will be derived from, each class with a virtual base must have a way to compute the location of the virtual base from the address point of its derived class.

In Visual C++, this is implemented by adding a hidden `vbptr` ("virtual base table pointer") field to each instance of a class with virtual bases. This field points to a shared, per-class table of displacements from the address point of the `vbptr` field to the class's virtual base(s).

Other implementations use embedded pointers from the derived class to its virtual bases, one per base. This other representation has the advantage of a smaller code sequence to address the virtual base, although an optimizing code generator can often common-subexpression-eliminate repeated virtual base access computations. However, it also has the disadvantages of larger instance sizes for classes with multiple virtual bases, of slower access to virtual bases of virtual bases (unless one incurs yet further hidden pointers), and of a less regular pointer to member dereference.

In Visual C++, `G` has a hidden `vbptr` which addresses a virtual base table whose second entry is `GdGvbptrC`. (This is our notation for "in `G`, the displacement from `G`'s `vbptr` to `C`". (We omit the prefix to "d" if the quantity is constant in all derived classes.)) For example, on a 32-bit platform, `GdGvbptrC` would be 8 (bytes). Similarly, the embedded `G` instance within an `I` addresses a `vbtable` customized for `G`'s within `I`'s, and so `IdGvbptrC` would be 20.

As can be seen from the layouts of `G`, `H`, and `I`, Visual C++ lays out classes with virtual bases by:

- Placing embedded instances of the non-virtually inherited bases first,
- Adding a hidden `vbptr` unless a suitable one was inherited from one of the non-virtual bases,
- Placing the new data members declared in the derived class, and, finally,
- Placing a single instance of each of the virtually inherited bases at the end of the instance.

This representation lets the virtually inherited bases "float" within the derived class (and its further derived classes) while keeping together and at constant relative displacements those parts of the object that are not virtual bases.

Data Member Access

Now that we have seen how classes are laid out, let's consider the cost to access data members of these classes.

No inheritance. In absence of inheritance, data member access is the same as in C: a dereference off some displacement from the pointer to the object.

```
C* pc;


pc->c1; // *(pc + dCc1);
```

Single inheritance. Since the displacement from the derived object to its embedded base instance is a constant 0, that constant 0 can be folded with the constant offset of the member within that base.

```
D* pd;
pd->c1; // *(pd + dDC + dCc1); // *(pd + dDCc1);
pd->d1; // *(pd + dDd1);
```

Multiple inheritance. Although the displacement to a given base, or to a base of a base, and so on, might be non-zero, it is still constant, and so any set of such displacements can be folded together into one constant displacement off the object pointer. Thus even with multiple inheritance, access to any member is inexpensive.

```
F* pf;
pf->c1; // *(pf + dFC + dCc1); // *(pf + dFc1);
pf->e1; // *(pf + dFE + dEe1); // *(pf + dFe1);
pf->f1; // *(pf + dFf1);
```

Virtual inheritance. Within a class with virtual bases, access to a data member or non-virtually inherited base class is again just a constant displacement off the object pointer. However, access to a data member of a virtual base is comparatively expensive, since it is necessary to fetch the `vbptr`, fetch a `vbtable` entry, and then add that displacement to the `vbptr` address point, just to compute the address of the data member. However, as shown for `i.c1` below, if the type of the derived class is statically known, the layout is also known, and it is unnecessary to load a `vbtable` entry to find the displacement to the virtual base.

```
I* pi;
pi->c1; // *(pi + dIGvbptr + (*(pi+dIGvbptr))[1] + dCc1);
pi->g1; // *(pi + dIG + dGg1); // *(pi + dIg1);
pi->h1; // *(pi + dIH + dHh1); // *(pi + dIh1);
pi->i1; // *(pi + dIi1);
I i;
i.c1; // *(&i + IdIC + dCc1); // *(&i + IdIc1);
```

What about access to members of transitive virtual bases, for example, members of virtual bases of virtual bases (and so on)? Some implementations follow one embedded virtual base pointer to the intermediate virtual base, then follow its virtual base pointer to its virtual base, and so on. Visual C++ optimizes such access by using additional `vbtable` entries which provide displacements from the derived class to any transitive virtual bases.

Casts

Except for classes with virtual bases, it is relatively inexpensive to explicitly cast a pointer into another pointer type. If there is a base-derived relationship between class pointers, the compiler simply adds or subtracts the displacement between the two (often 0).

```
F* pf;
(C*)pf; // (C*)(pf ? pf + dFC : 0); // (C*)pf;
(E*)pf; // (E*)(pf ? pf + dFE : 0);
```

In the `C*` cast, no computations are required, because `dFC` is 0. In the `E*` cast, we must add `dFE`, a non-zero constant, to the pointer. C++ requires that null pointers (0) remain null after a cast. Therefore Visual C++ checks for null before performing the addition. This check occurs only when a pointer is implicitly or explicitly converted to a related pointer type, not when a `derived*` is implicitly converted to a `base*const this` pointer when a base member function is invoked on a derived object.

As you might expect, casting over a virtual inheritance path is relatively expensive: about the same cost as accessing a member of a virtual base:

```
I* pi;
(G*)pi; // (G*)pi;
(H*)pi; // (H*)(pi ? pi + dIH : 0);
(C*)pi; // (C*)(pi ? (pi+dIGvbptr + (*(pi+dIGvbptr))[1]) : 0);
```

In general, you can avoid a lot of expensive virtual base field accesses by replacing them with one cast to the virtual base and base relative accesses:

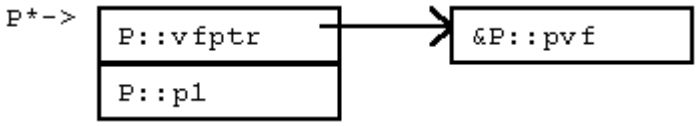
```
/* before: */             ... pi->c1 ... pi->c1 ...
/* faster: */ C* pc = pi; ... pc->c1 ... pc->c1 ...
```

Member Functions

A C++ member function is just another member in the scope of its class. Each (non-static) member function of a class `X` receives a special hidden `this` parameter of type `X *const`, which is implicitly initialized from the object the member function is applied to. Also, within the body of a member function, member access off the `this` pointer is implicit.

```
struct P {
   int p1;
   void pf(); // new
   virtual void pvf(); // new


};
```



`P` has a non-virtual member function `pf()` and a virtual member function `pvf()`. It is apparent that virtual member functions incur an instance size hit, as they require a virtual function table pointer. More on that later. Notice there is no instance cost to declaring non-virtual member functions. Now consider the definition of `P::pf()`:

```
void P::pf() { // void P::pf([P *const this])
   ++p1;   // ++(this->p1);
}
```

Here `P::pf()` receives a hidden `this` parameter, which the compiler has to pass each call. Also note that member access can be more expensive than it looks, because member accesses are `this` relative. On the other hand, compilers commonly enregister `this` so member access cost is often no worse than accessing a local variable. On the other hand, compilers may not be able to enregister the instance data itself because of the possibility `this` is aliased with some other data.

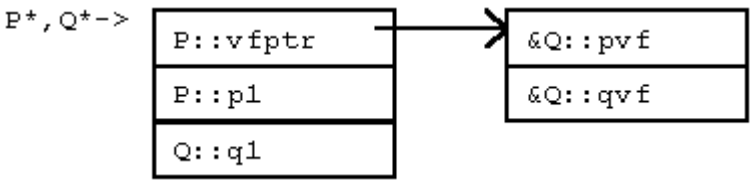
Overriding Member Functions

Member functions are inherited just as data members are. Unlike data members, a derived class can override, or replace, the actual function definition to be used when an inherited member function is applied to a derived instance. Whether the override is static (determined at compile time by the static types involved in the member function call) or dynamic (determined at run-time by the dynamic object addressed by the object pointer) depends upon whether the member function is declared `virtual`.

Class `Q` inherits `P`'s data and function members. It declares `pf()`, overriding `P::pf()`. It also declares `pvf()`, a virtual function overriding `P::pvf()`, and declares a new non-virtual member function `qf()`, and a new virtual function `qvf()`.

```
struct Q : P {
   int q1;
   void pf();  // overrides P::pf
   void qf();  // new
   void pvf(); // overrides P::pvf
   virtual void qvf(); // new


};
```



For non-virtual function calls, the member function to call is statically determined, at compile time, by the type of the pointer expression to the left of the `->` operator. In particular, even though `ppq` points to an instance of `Q`, `ppq->pf()` calls `P::pf()`. (Also notice the pointer expression left of the `->` is passed as the hidden `this` parameter.)

```
P p; P* pp = &p; Q q; P* ppq = &q; Q* pq = &q;
pp->pf();  // pp->P::pf();  // P::pf(pp);
ppq->pf(); // ppq->P::pf(); // P::pf(ppq);
pq->pf();  // pq->Q::pf();  // Q::pf((P*)pq);
pq->qf();  // pq->Q::qf();  // Q::qf(pq);
```

For virtual function calls, the member function to call is determined at run-time. Regardless of the declared type of the pointer expression left of the `->` operator, the virtual function to call is the one appropriate to the type of the actual instance addressed by the pointer. In particular, although `ppq` has type `P*`, it addresses a `Q`, and so `Q::pvf()` is called.

```
pp->pvf();  // pp->P::pvf();  // P::pvf(pp);
ppq->pvf(); // ppq->Q::pvf(); // Q::pvf((Q*)ppq);
pq->pvf();  // pq->Q::pvf();  // Q::pvf((P*)pq);
```

Hidden `vfptr` members are introduced to implement this mechanism. A `vfptr` is added to a class (if it doesn't already have one) to address that class's virtual function table (`vftable`). Each virtual function in a class has a corresponding entry in that class's `vftable`. Each entry holds the address of the virtual function override appropriate to that class. Therefore, calling a virtual function requires fetching the instance's `vfptr`, and indirectly calling through one of the `vftable` entries addressed by that pointer. This is in addition to the usual function call overhead of parameter passing, call, and return instructions. In the example below, we fetch `q`'s `vfptr`, which addresses `Q`'s `vftable`, whose first entry is `&Q::pvf`. Thus `Q::pvf()` is called.

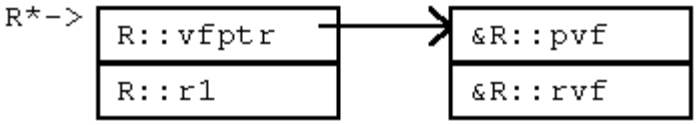
Looking back at the layouts of `P` and `Q`, we see that the Visual C++ compiler has placed the hidden `vfptr` member at the start of the `P` and `Q` instances. This helps ensure that virtual function dispatch is as fast as possible. In fact, the Visual C++ implementation *ensures* that the first field in any class with virtual functions is always a `vfptr`. This can require inserting the new `vfptr` before base classes in the instance layout, or even require that a right base class that does begin with a `vfptr` be placed before a left base that does not have one.

Most C++ implementations will share or reuse an inherited base's `vfptr`. Here `Q` did not receive an additional `vfptr` to address a table for its new virtual function `qvf()`. Instead, a `qvf` entry is appended to the end of `P`'s `vftable` layout. In this way, single inheritance remains inexpensive. Once an instance has a `vfptr` it doesn't need another one. New derived classes can introduce yet more virtual functions, and their `vftable` entries are simply appended to the end of their one per-class `vftable`.

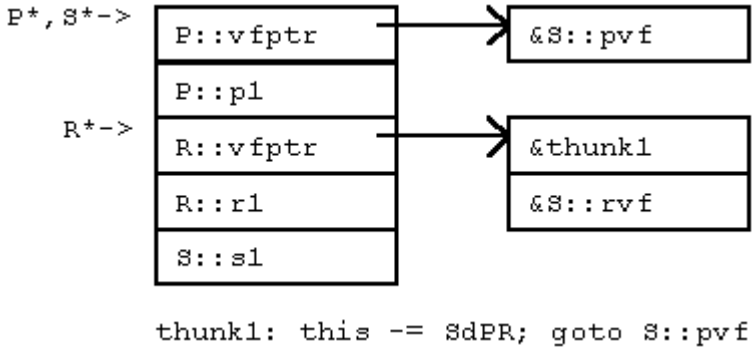
Virtual Functions: Multiple Inheritance

It is possible for an instance to contain more than one `vfptr` if it inherits them from multiple bases, each with virtual functions. Consider `R` and `S`:

```
struct R {
   int r1;
   virtual void pvf(); // new
   virtual void rvf(); // new
};
```




```
struct S : P, R {
   int s1;
   void pvf(); // overrides P::pvf and R::pvf
   void rvf(); // overrides R::rvf
   void svf(); // new
};
```



Here `R` is just another class with some virtual functions. Since `S` multiply inherits, from `P` and `R`, it contains an embedded instance of each, plus its own instance data contribution, `S::s1`. Notice the right base `R` has a different address point than do `P` and `S`, as expected with multiple inheritance. `S::pvf()` overrides both `P::pvf()` and `R::pvf()`, and `S::rvf()` overrides `R::rvf()`. Here are the required semantics for the `pvf` override:

```
S s; S* ps = &s;
((P*)ps)->pvf(); // ((P*)ps)->P::vfptr[0])((S*)(P*)ps)
((R*)ps)->pvf(); // ((R*)ps)->R::vfptr[0])((S*)(R*)ps)
ps->pvf();       // one of the above; calls S::pvf()
```

Since `S::pvf()` overrides both `P::pvf()` and `R::pvf()`, it must replace their `vftable` entries in the `S` `vftable`s. However, notice that it is possible to invoke `pvf()` both as a `P` and an `R`. The problem is that `R`'s address point does not correspond to `P`'s and `S`'s. The expression `(R*)ps` does not point to the same part of the class as does `(P*)ps`. Since the function `S::pvf()` expects to receive an `S*` as its hidden `this` parameter, the virtual function call itself must automatically convert the `R*` at the call site into an `S*` at the callee. Therefore, `S`'s copy of `R`'s `vftable`'s `pvf` slot takes the address of an adjuster thunk, which applies the address adjustment necessary to convert an `R*` pointer into an `S*` as desired.

In MSC++, for multiple inheritance with virtual functions, adjuster thunks are required only when a derived class virtual function overrides virtual functions of multiple base classes.

Address Points and "Logical This Adjustment"

Consider next `S::rvf()`, which overrides `R::rvf()`. Most implementations note that `S::rvf()` must have a hidden `this` parameter of type `S*`. Since `R`'s `rvf` `vftable` slot may be used when this call occurs:

```
((R*)ps)->rvf(); // (*((R*)ps)->R::vfptr[1])((R*)ps)
```

Most implementations add another thunk to convert the `R*` passed to `rvf` into an `S*`. Some also add an additional `vftable` entry to the end of `S`'s `vftable` to provide a way to call `ps->rvf()` without first converting to an `R*`. MSC++ avoids this by intentionally compiling `S::rvf()` so as to expect a `this` pointer which addresses not the `S` object but rather the `R` embedded instance within the `S`. (We call this “giving overrides the same expected address point as in the class that first introduced this virtual function”.) This is all done transparently, by applying a “logical this adjustment” to all member fetches, conversions from `this`, and so on, that occur within the member function. (Just as with multiple inheritance member access, this adjustment is constant-folded into other member displacement address arithmetic.)

Of course, we have to compensate for this adjustment in our debugger.

```
ps->rvf(); // ((R*)ps)->rvf(); // S::rvf((R*)ps)
```

Thus MSC++ generally avoids creating a thunk and an additional extra `vftable` entry when overriding virtual functions of non-leftmost bases.

Adjuster Thunks

As described, an adjuster thunk is sometimes called for, to adjust `this` (which is found just below the return address on the stack, or in a register) by some constant displacement en route to the called virtual function. Some implementations (especially cfront-based ones) do not employ adjuster thunks. Rather, they add additional displacement fields to each virtual function table entry. Whenever a virtual function is called, the displacement field, which is quite often 0, is added to the object address as it is passed in to become the `this` pointer:

```
ps->rvf();
// struct { void (*pfn)(void*); size_t disp; };
// (*ps->vfptr[i].pfn)(ps + ps->vfptr[i].disp);
```

The disadvantages of this approach include both larger `vftable`s and larger code sequences to call virtual functions.

More modern PC-based implementations use adjust-and-jump techniques:

```
S::pvf-adjust: // MSC++
   this -= SdPR;
   goto S::pvf()
```

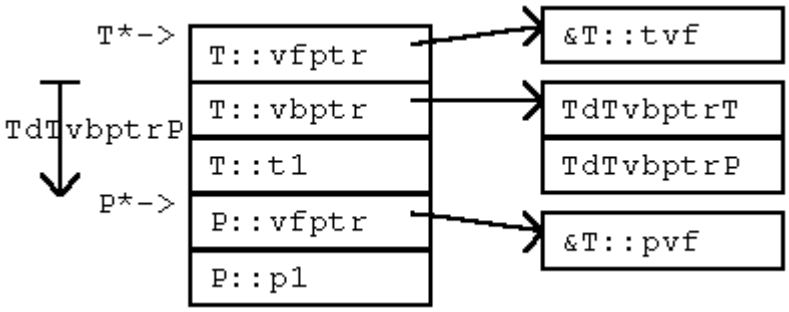
Of course, the following code sequence is even better (but no current implementation generates it):

```
S::pvf-adjust:
   this -= SdPR; // fall into S::pvf()
S::pvf() { ... }
```

Virtual Functions: Virtual Inheritance

Here `T` virtually inherits `P` and overrides some of its member functions. In Visual C++, to avoid costly conversions to the virtual base `P` when fetching a `vftable` entry, new virtual functions of `T` receive entries in a new `vftable`, requiring a new `vfptr`, introduced at the top of `T`.

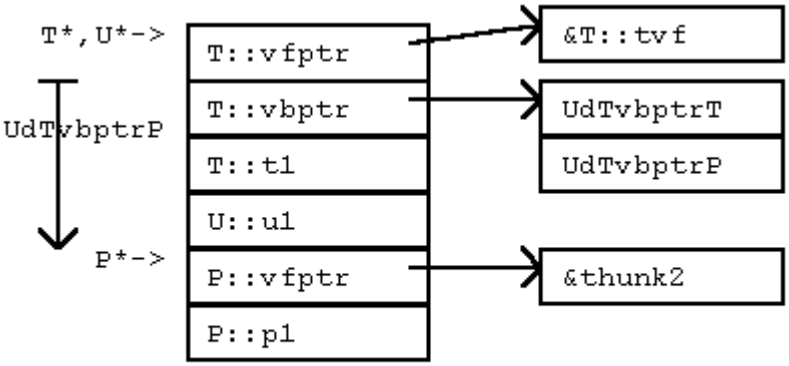
```
struct T : virtual P {
   int t1;
   void pvf();         // overrides P::pvf
   virtual void tvf(); // new
};
```




```
void T::pvf() {
   ++p1; // ((P*)this)->p1++; // vbtable lookup!
   ++t1; // this->t1++;
}
```

As shown above, even within the definition of a virtual function, access to data members of a virtual base must still use the `vbtable` to fetch a displacement to the virtual base. This is necessary because the virtual function can be subsequently inherited by a further derived class with different layout with respect to virtual base placement. And here is just such a class:

```
struct U : T {
   int u1;
};
```




```
thunk2: this -= (UdP - TdP); goto T::pvf
```

Here `U` adds another data member, which changes the `dP`, the displacement to `P`. Since `T::pvf` expects to be called with a `P*` in a `T`, an adjuster thunk is necessary to adjust `this` so it arrives at the callee addressing just past `T::t1` (the address point of a `P*` in a `T`). (Whew! That's about as complex as things get!)

Special Member Functions

This section examines hidden code compiled into (or around) your special member functions.

Constructors and Destructors

As we have seen, sometimes there are hidden members that need to be initialized during construction and destruction. Worst case, a constructor may perform these activities

- If "most-derived," initialize `vbptr` field(s) and call constructors of virtual bases.
- Call constructors of direct non-virtual base classes.
- Call constructors of data members.
- Initialize `vfptr` field(s).
- Perform user-specified initialization code in body of constructor definition.

(A "most-derived" instance is an instance that is not an embedded base instance within some other derived class.)

So, if you have a deep inheritance hierarchy, even a single inheritance one, construction of an object may require many successive initializations of a class's `vfptr`. (Where appropriate, Visual C++ will optimize away these redundant stores.)

Conversely, a destructor must tear down the object in the exact reverse order to how it

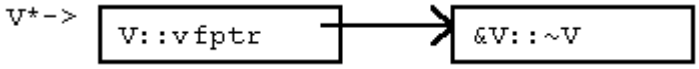
was initialized:

- Initialize `vfptr` field(s).
- Perform user-specified destruction code in body of destructor definition.
- Call destructors of data members (in reverse order).
- Call destructors of direct non-virtual bases (in reverse order).
- If “most-derived,” call destructors of virtual bases (in reverse order).

In Visual C++, constructors for classes with virtual bases receive a hidden “most-derived flag” to indicate whether or not virtual bases should be initialized. For destructors, we use a “layered destructor model,” so that one (hidden) destructor function is synthesized and called to destroy a class *including* its virtual bases (a “most-derived” instance) and another to destroy a class *excluding* its virtual bases. The former calls the latter, then destroys virtual bases (in reverse order).

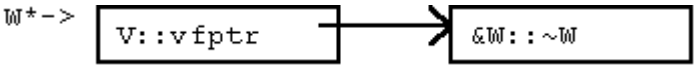
Virtual Destructors and Operator Delete

Consider structs `V` and `W`.




```
struct V {


   virtual ~V();
};
```




```
struct W : V {
   operator delete();
};
```

Destructors can be virtual. A class with a virtual destructor receives a hidden `vfptr` member, as usual, which addresses a `vftable`. The table contains an entry holding the address of the virtual destructor function appropriate for the class. What is special about virtual destructors is they are implicitly invoked when an instance of a class is deleted. The call site (delete site) does not know what the dynamic type being destroyed is, and yet it must invoke the appropriate operator delete for that type.

For instance, when `pv` below addresses a `W`, after `W::~W()` is called, its storage must be destroyed using `W::operator delete()`.

```
V* pv = new V;
delete pv;   // pv->~V::V(); // use ::operator delete()
pv = new W;
delete pv;   // pv->~W::W(); // use W::operator delete()
pv = new W;
::delete pv; // pv->~W::W(); // use ::operator delete()
```

To implement these semantics, Visual C++ extends its “layered destructor model” to automatically create another hidden destructor helper function, the “deleting destructor,” whose address replaces that of the “real” virtual destructor in the virtual function table. This function calls the destructor appropriate for the class, then optionally invokes the appropriate operator delete for the class.

Arrays

Dynamic (heap allocated) arrays further complicate the responsibilities of a virtual destructor. There are two sources of complexity. First, the dynamic size of a heap allocated array must be stored along with the array itself, so dynamically allocated arrays automatically allocate extra storage to hold the number of array elements. The other complication occurs because a derived class may be larger than a base class, yet it is imperative that an array delete correctly destroy each array element, even in contexts where the array size is not evident:

```
struct WW : W { int w1; };
pv = new W[m];
delete [] pv; // delete m W's  (sizeof(W)  == sizeof(V))
pv = new WW[n];
delete [] pv; // delete n WW's (sizeof(WW) >  sizeof(V))
```

Although, strictly speaking, polymorphic array delete is undefined behavior, we had several customer requests to implement it anyway. Therefore, in MSC++, this is implemented by yet another synthesized virtual destructor helper function, the so-called “vector delete destructor,” which (since it is customized for a particular class, such as `WW`) has no difficulty iterating through the array elements (in reverse order), calling the appropriate destructor for each.

Exception Handling

Briefly, the exception handling proposal in the C++ standards committee working papers provides a facility by which a function can notify its callers of an exceptional condition and select appropriate code to deal with the situation. This provides an alternative mechanism to the conventional method of checking error status return codes at every function call return site.

Since C++ is object-oriented, it should come as no surprise that objects are employed to represent the exception state, and that the appropriate exception handler is selected based upon the static or dynamic type of exception object “thrown.” Also, since C++ always ensures that frame objects that are going out of scope are properly destroyed, implementations must ensure that in transferring control (unwinding the stack frame) from throw site to “catch” site, (automatic) frame objects are properly destroyed.

Consider this example:

```
struct X { X(); }; // exception object class
struct Z { Z(); ~Z(); }; // class with a destructor
extern void recover(const X&);
void f(int), g(int);

int main() {
   try {
      f(0);
   } catch (const X& rx) {
      recover(rx);
   }
   return 0;
}

void f(int i) {
   Z z1;
   g(i);
   Z z2;
   g(i-1);
}

void g(int j) {
   if (j < 0)
      throw X();
}
```

This program will throw an exception. `main()` establishes an exception handler context for its call to `f(0)`, which in turn constructs `z1`, calls `g(0)`, constructs `z2`, and calls `g(-1)`. `g()` detects the negative argument condition and throws an `X` object exception to whatever caller can handle it. Since neither `g()` nor `f()` established an exception handler context, we consider whether the exception handler established by `main()` can handle an `X` object

exception. Indeed it can. Before control is transferred to the catch clause in `main()`, however, objects on the frame between the throw site in `g()` and the catch site in `main()` must be destroyed. In this case, `z2` and `z1` are therefore destroyed.

An exception handling implementation might employ tables at the throw site and the catch site to describe the set of types that might catch the thrown object (in general) and can catch the thrown object at this specific catch site, respectively, and generally, how the thrown object should initialize the catch clause "actual parameter." Reasonable encoding choices can ensure that these tables do not occupy too much space.

However, let us reconsider function `f()`. It looks innocuous enough. Certainly it contains neither `try`, `catch`, nor `throw` keywords, so exception handling would not appear to have much of an impact on `f()`. Wrong! The compiler must ensure that, once `z1` is constructed, if any subsequently called function were to raise an exception ("throw") back to `f()`, and therefore out of `f()`, that the `z1` object is properly destroyed. Similarly, once `z2` is constructed, it must ensure that a subsequent throw is sure to destroy `z2` and then `z1`.

To implement these "unwind semantics," an implementation must, behind the scenes, provide a mechanism to dynamically determine the context (site), in a caller function, of the call that is raising the exception. This can involve additional code in each function prolog and epilog, and, worse, updates of state variables between each set of object initializations. For instance, in the example above, the context in which `z1` should be destroyed is clearly distinct from the subsequent context in which `z2` and then `z1` should be destroyed, and therefore Visual C++ updates (stores) a new value in a state variable after construction of `z1` and again after construction of `z2`.

All these tables, function prologs, epilogs, and state variable updates, can make exception handling functionality a significant space and speed expense. As we have seen, this expense is incurred even in functions that do not employ exception handling constructs.

Fortunately, some compilers provide a compilation switch and other mechanisms to disable exception handling and its overhead from code that does not require it.

Summary

There, now go write your own compiler.

Seriously, we have considered many of the significant C++ run-time implementation issues. We see that some wonderful language features are almost free, and others can incur significant overhead. These implementation mechanisms are applied quietly for you, behind the curtains, so to speak, and it is often hard to tell what a piece of code costs when looking at it in isolation. The frugal coder is well advised to study the generated native code from time to time and question whether use of this or that particularly cool language feature is worth its overhead.

Acknowledgments. The Microsoft C++ Object Model described herein was originally designed by Martin O'Riordan and David Jones; yours truly added details here and there as necessary to complete the implementation.

PDF version was generated on January 3, 2006